**C**

**Preset:**

* Invented by Dennis Ritchie, 1972, Bell Labs to write Unix OS
* General Purpose, modular and portable
* Low-level memory access
* C-Standards: C89/90 (1989/90), C99 (1999), C11 (2011), C18(2018)

**Structure of C Program:**

|  |  |
| --- | --- |
| Header | #include <stdio.h> |
| main() | int main() { |
| Variable Declaration | int a = 10; |
| Body | printf(“%d”, a”); |
| Return | return 0;  } |

* Header Files inclusion: Contains C function declarations and macro definitions to be shared between source files.
* Main Method: Execution starts from this function.
* Variable Declaration: Variables that are to be used in the function. No variables can be used without being declared.
* Body: Refers to operations that are performed in the functions.
* Return: Depending on return type of the function, the return refers to the returning values from a function

**Memory Map:**

**A diagram of a stack

Description automatically generated**

**Compilation:** Process of converting the source code of C lang into machine code.

A diagram of a computer program

Description automatically generated

1. Source File: Create a c program using an editor and save the file as filename.c
2. Pre-processor: This phase includes the following and produces filename.i
   1. Removal of comments
   2. Expansion of Macros
   3. Expansion of Included files
   4. Conditional Compilation
3. Compiling: Compiles filename.i and produces filename.s which includes assembly-level instructions.
4. Assembler: Takes filename.s as output and produces filename.o – object code. Existing code is converted into machine language.
5. Linking: All function calls linked, adds extra code to mark delimiters of each block and forms a executable file.
6. Loading: Loading the process to main memory

*$gcc -Wall -save-temps filename.c –o* *filename* // To save all intermediate files during compilation.

**Comments and Line Splicing:**

Single Line: //

Multi Line: /\* \*/

*A screenshot of a computer code

Description automatically generated*

*Prefer using Multi Line Comments everywhere.*

**Data Types:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Bits** | **Range** |
| char | 8 | -127 to 127 |
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | -127 to 127 |
| int | 16/32 | -32767 to 32767 |
| unsigned int | 16/32 | 0 to 65535 |
| signed int | 16/32 | -32767 to 32767 |
| short int | 16 | -32767 to 32767 |
| unsigned short int | 16 | 0 to 65535 |
| signed short int | 16 | -32767 to 32767 |
| long int | 32 | -231-1 to 231 |
| long long int | 64 | -263-1 to 263 |
| signed long int | 32 | -231-1 to 231 |
| unsigned long int | 32 | 0 to 231 |
| unsigned long long int | 64 | 264 - 1 |
| float | 32 | 0 to 231 |
| double | 64 | 264 - 1 |
| long double | 80 | 280 – 1 |
| bool |  | stdbool.h |
| float\_complex |  | complex.h |
| float\_imaginary |  | complex.h |
| double\_complex |  | complex.h |
| double\_imaginary |  | complex.h |
| long double\_complex |  | complex.h |
| long double\_imaginary |  | complex.h |

INT\_MIN and INT\_MAX in limits.h give the limits of an integer

**Operators:**

|  |  |
| --- | --- |
| Assignment | =, +=, -=, \*=, /=, %=, &=, |=, ^=, >>=, <<= |
| Arithmetic | + , - , \* , / , % , ++ , -- |
| Relational | ==, !=, >, <, >=, <= |
| Logical | &&, ||, ! |
| Bitwise | &, |, ^ |
| Ternary | Condition? True : False; |
| Pointer | &, \* |
| Misc | sizeof(), . (obj.val), &(ptr), \*(access) |

**Identifiers - Variables and Constants:**

* Named location which has some memory allocated to it.
* Always start with letter/\_, only alphanumeric, no whitespace, no keywords.

**Keywords:** Predefined and reserved words that special meanings to compiler

*A grid of words

Description automatically generated*

**Type Qualifiers:**

|  |  |
| --- | --- |
| const | Variables of type const cannot be changed by program. They can only be given an initial value. |
| volatile | Variable's value may be changed in ways not explicitly specified by the program. Global variable which is passed to the OS's clock routine is a good example. |
| restrict | To specify that a pointer is an exclusive method of accessing a specific object or area of memory. |
| \_Atomic | A variable that may be read or written atomically - without interference of other threads or processes |

**Storage Class Specifiers:**

|  |  |
| --- | --- |
| auto | Default storage class for all the variables declared inside a function or a block |
| extern | Used to specify that an object is declared with external linkage elsewhere in the program. |
| static | Permanent variables within their own function or file. They maintain their values between function calls. |
| register | Variables declared with register are stored in CPU registers instead of RAM for faster access. |

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

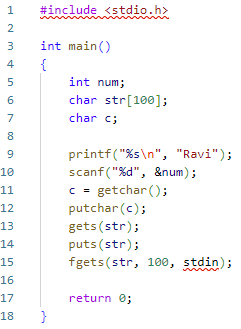
A screenshot of a computer program

Description automatically generated  
 **

**Scope Rules:**

|  |  |
| --- | --- |
| **Scope** | **Meaning** |
| File Scope | Starts at beginning of the file and ends with end of the file |
| Block Scope | Starts at the opening of { and ends at } |
| Function prototype scope | Identifiers declared in a func prototype; visible within the prototype |
| Function Scope | Starts at the opening of { of a function and ends at } of a function |

**Input/Output:**

****

* scanf() stops reading characters when it encounters a space
* gets() reads a line of text from stdin(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs.
* puts() writes the string str with a newline character ('\n') at the end to stdout.
* fgets reads characters and stores them as a C string into str until 100 characters have been read or either a newline or the end-of-file is reached, whichever happens first.
* gets() will keep reading until it encounters a newline character. Unless the buffer is large enough, or the length of the line being read is known ahead of time, gets() can potentially overflow the input buffer and start overwriting memory it is not supposed to, wreaking havoc or opening security vulnerabilities.
* getch(): Reads a char without echo; does not wait for carriage return
* getche(): Reads a char with echo; does not wait for carriage return

**Format Specifiers:**

|  |  |
| --- | --- |
| int | %d, %i |
| char | %c |
| float | %f |
| double | %lf |
| short int | %hd |
| unsigned int | %u |
| long int | %li |
| long long int | %lli |
| unsigned long int | %lu |
| unsigned long long int | %llu |
| signed char | %c |
| unsigned char | %c |
| long double | %Lf |

**Escape Sequences:**

|  |  |
| --- | --- |
| \a | Alarm or Beep |
| \b | Backspace |
| \f | Form Feed |
| \n | New Line |
| \r | Carriage Return |
| \t | Tab (Horizontal) |
| \v | Tab (Vertical) |
| \\ | Backslash |
| \’ | Single Quote |
| \” | Double Quote |
| \? | Question Mark |
| \ooo | Octal Number |
| \xhh | Hexadecimal number |
| \0 | Null |

**Control Statements:**

|  |  |
| --- | --- |
| Conditional | if-else-if |
|  | switch |
| Loops | for |
|  | while |
|  | do-while |
| Jump | return |
|  | goto |
|  | break |
|  | exit() |
|  | continue |
| Expression | general statement with ; |

**A screenshot of a computer

Description automatically generated**

**Macros and Preprocessors:**

All lines that start with **#** are processed by preprocessor; the functionality is processed prior to other statements in the program.

* When we use *include* directive, the contents of included header file are copied to the current file. Angulars **<** and **>** indicate preprocessor to look in the standard folder where all header files are held. Double Quotes **“** and **“** indicate it to look into the current folder.
* When we use *define* for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression.
* Macros can take full function like args and args are not checked for data type.



* Macro args are not evaluated before macro expansion.

A white background with colorful text

Description automatically generated

* Tokens passed to macros can be concatenated using **##**



* A token passed to macro can be converted to a string literal by using # before it



* The macros can be written in multiple lines:

A computer code on a white background

Description automatically generated

* It’s better to avoid macros with arguments and inline functions should be preferred as they have type checking.
* Preprocessors also support if-else directives: #if, #else, #endif, #ifdef, #ifndef, #undef
* Standard Macros:

*A screen shot of a computer code

Description automatically generated*

**void main() or main():** In C, void main() has no defined(legit) usage, and it can sometimes throw garbage results or an error. However, main() is used to denote the main function which takes no arguments and returns an integer data type. To summarize the above, it is never a good idea to use void main() or simply, main() as it doesn’t confirm standards. It may be allowed by some compilers though.

**int main() and int main(void):** In C, if a function signature doesn’t specify any argument, it means that the function can be called with any number of parameters or without any parameters.

**Command Line Arguments:** A command line argument is the info that follows the program's name on the command line of the OS. Two built-in arguments: - argc holds the number of arguments on the command line and is an integer. - argv is pointer to an array of character pointers. Each element in this array points to a command line argument.

int main(int argc, char \*argv[])

**Call by Value:** This method of passing args to a subroutine copies the values of args into the formal parameters of the sub routine. Changes made to the parameter have no effect on the arg.

**Call by Reference:** This method of passing args to a subroutine copies the addresses of args into parameters. Inside the subroutine, the address is used to access the actual arg. Changes made to the parameter affect the arg.

**Program Error Signals:**

|  |  |  |
| --- | --- | --- |
| SIGFPE | Arithmetic Error | Division by zero, Floating point error etc. |
| SIGILL | Illegal Instruction: Instruction with no privilege to get executed, got executed. | Stack overflow, Object file corrupted |
| SIGSEGV | Segmentation fault: Process trying to access a mem location not allocated to it. | De-referencing a wild pointer, Programs gets far from its mem space |
| SIGBUS | Bus error. Invalid memory is accessed (not existing mem accessed) | De-referencing memory location out of mem space |
| SIGABRT | Internal Error | General used in assert function |
| SIGSYS | System Call error | Invalid arg passed to a sys call |
| SIGTRAP | Exception Occurred, debugger to be informed | Variable changes its value. |

**Files:**

**A screenshot of a computer program

Description automatically generated**

**Pointers:**

* A Pointer is a variable that holds a memory address of another object in memory.
* Pointer Arithmetic: Incrementing a pointer makes it point to a memory location which equals current memory location + sizeof(Pointer data type).
* Function Pointers: Pointers having memory address of where the function begins

A screenshot of a computer program

Description automatically generated

**Dynamic Allocation:**

• malloc(): Allocates the required memory space during execution time. Returns address of the first byte of allocated space/NULL if the memory allocation fails

Syntax: ptr = (datatype \*) malloc (size); //stdlib.h

• calloc(): Allocates required memory size during execution time and initializes memory with 0s. Returns address of the first byte of allocated space/NULL if the memory allocation fails.

Syntax: ptr = (datatype \*) calloc (n, size); //stdlib.h; n= no. of blocks

• realloc(): To reallocate the size of memory allocated by malloc() or calloc()

Syntax: ptr = (datatype \*) realloc (ptr, size); //stdlib.h

• free(): De-allocates the memory allocated by malloc() or calloc()

free(ptr);

**Arrays:**

Single Dimension Arrays: type var\_name[size];

double balance[100];

Two Dimensional Arrays:

int d[10][20]; // first - rows, second, columns

Multidimensional Arrays:

Syntax: type name[size 1][size 2]….[size n];

int m[4][3][4][5];

**Structures:**

Declaration:

struct student {

char name[30];

int age;

};

typedef struct student STUDENT;

Access:

STUDENT ram;

ram.name = “Ram”;

ram.age = 5;

**Unions:**

union student {

char name[30];

int age;

};

// Only member is active at a time.

**Enumerations:**

A screenshot of a computer program

Description automatically generated